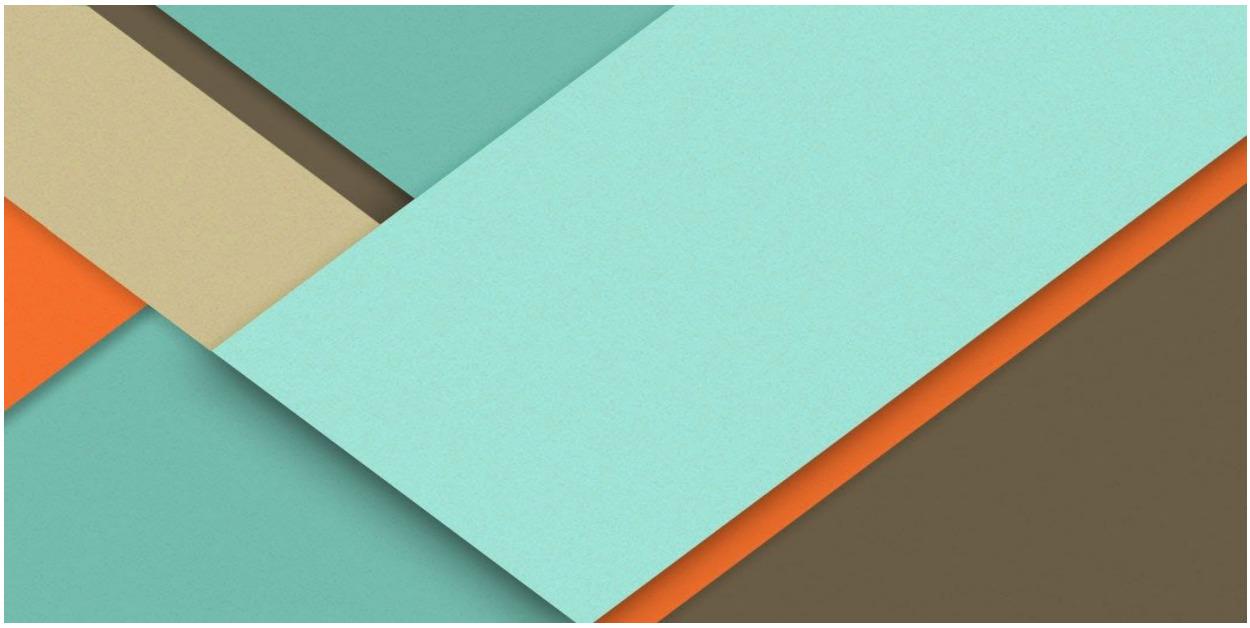

Compiler Project | Phase 2

27.04.2020

Ahmed Nabil (9)
Abdelrahman Alsayed Ahmed (24)
Mohamed Samy (40)
Yehia Elsayed (62)

Overview

This phase of the assignment aims to practice techniques for building automatic parser generator tools. The parser generator expects an LL (1) grammar as input. It should compute First and Follow sets and uses them to construct a predictive parsing table for the grammar.

Description of used Data Structures

- Map <String, Rule>: To store each non-terminal and pointer to its corresponding Rule.
- Vector<String>: To store the productions of each Rule as strings to be used later.
- vector<vector<string>>: To store the entries of the predictive table.
- unorderd_map<string,int>: To store each non-terminal and each terminal name and index in the predictive table.
- Classes to store the required information about each rule to improve the running time and applying the OOP design concepts.

Algorithms And Techniques

- ### Remove Left Recursion :

  Using the algorithm of replacing each non-terminal from the last rules in current rules by its productions. And checking if it has immediate left recursion if so it will remove the left recursion according to the lectures by building the new rule for the new non-terminal and add it to the map of the rules.

- ### Left factoring :

  We have sorted vector<string> caring the productions of each rule then splitting each production by the space in a vector<vector<string>> in which each row represent a vector<string> of splitted production, then we have to check in each section of the productions beginning with the same letter for the first components of each production that is common along biggest no of productions recursively, then adding the new rule for the common factor if found.

- ### Compute first :

  Compute first of each rule recursively as there is no left recursion by looping throw every production of the rule and check for the first component of the production
  - If it is terminal: add it to the first of the rule
  - If it is nonterminal: solve it recursively and add its first to the first of the current non-terminal.

- ### Compute Follow :

  Using DFS and backtracking algorithms calculate the following set of each non-terminal by using the first sets in case of needing it. Adding $ sign to the follow set of the first non-terminal and continue with the algorithm computing the rest of the follow sets as discussed in lectures.

- ### Build the predictive table :

  We have used the algorithm of iterating through each rule with its name(Non-terminal symbol) and checking for the first terminal symbols of each production to fill its entry in the table with the production rule also looking for the epsilon in each first set and if it is found we also add the production rule in the entry of the rule and its follow terminal symbol.
  At the end if there is no epsilon in the first set of each production rule in the rule we use its entries with the follow terminal symbols as synch tokens.
  Any other undefined entries are considered as errors.

- ### Stack procedure and error handling :

  We have used the general algorithm for accepting the inputs of tokens. We have used mainly stack and vectors of tokens. Also, we handled different types of errors(Terminals are not identical - Existence of “Error” in the predicative table) by taking the suitable action that is mentioned in the lecture under the headline(Panic-Mode Error Recovery).

Predictive Table

https://drive.google.com/open?id=1BxnKjMNiGTqJ7yYdCZh82F5AWSu1pDA-

Parsing Process

https://drive.google.com/open?id=1cLRT7uVqxxUUZHDvbRpGtxxs995JbsKy

Explanation of functions of phases

After applying the lexical rules to the input then we start to use the tokens generated by the lexical analyzer as the input of the parser table token by token and use the stack procedure as we explained in the algorithms part.

Assumptions and justification

- Since we have to split the rules by the " = " sign and some rules have " = " among its production we treat the " = " as " assign " word when building the predictive table.

Test Case Screenshots

Rules:

```
# METHOD_BODY = STATEMENT_LIST
# STATEMENT_LIST = STATEMENT | STATEMENT_LIST STATEMENT
# STATEMENT = DECLARATION
| IF
| WHILE
| ASSIGNMENT
# DECLARATION = PRIMITIVE_TYPE 'id' ';'
# PRIMITIVE_TYPE = 'int' | 'float'
# IF = 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'
# WHILE = 'while' '(' EXPRESSION ')' '{' STATEMENT '}'
# ASSIGNMENT = 'id' '=' EXPRESSION ';'
# EXPRESSION = SIMPLE_EXPRESSION
| SIMPLE_EXPRESSION 'relop' SIMPLE_EXPRESSION
# SIMPLE_EXPRESSION = TERM | SIGN TERM | SIMPLE_EXPRESSION 'addop' TERM
# TERM = FACTOR | TERM 'mulop' FACTOR
# FACTOR = 'id' | 'num' | '(' EXPRESSION ')'
# SIGN = '+' | '-'
```

Input Code :

```
int x;
x = 5;
if (x > 2)
{
 x = 0;
}
```

The output of the lexical:

```
int
id
;
id
assign
num
;
if
(
id
relop
num
)
{
id
assign
num
;
}
```

The output of the Parser:

```
METHOD_BODY
STATEMENT_LIST
'int' 'id' ';' STATEMENT_LIST`
'int' 'id' ';' STATEMENT STATEMENT_LIST`
'int' 'id' ';' 'id' '=' EXPRESSION ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' SIMPLE_EXPRESSION EXPRESSION` ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' TERM SIMPLE_EXPRESSION` EXPRESSION` ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num' TERM` SIMPLE_EXPRESSION` EXPRESSION` ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'  SIMPLE_EXPRESSION` EXPRESSION` ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'   EXPRESSION` ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' STATEMENT STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' SIMPLE_EXPRESSION EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' TERM SIMPLE_EXPRESSION` EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id' TERM` SIMPLE_EXPRESSION` EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'  SIMPLE_EXPRESSION` EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'   EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' SIMPLE_EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' TERM SIMPLE_EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num' TERM` SIMPLE_EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'  SIMPLE_EXPRESSION` ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' EXPRESSION ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' SIMPLE_EXPRESSION EXPRESSION` ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' TERM SIMPLE_EXPRESSION` EXPRESSION` ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' 'num' TERM` SIMPLE_EXPRESSION` EXPRESSION` ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' 'num'  SIMPLE_EXPRESSION` EXPRESSION` ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' 'num'   EXPRESSION` ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' 'num'    ';' '}' 'else' '{' STATEMENT '}' STATEMENT_LIST`
Error: missing 'else'
Error: missing '{'
Error: missing '}'
'int' 'id' ';' 'id' '=' 'num'    ';' 'if' '(' 'id'    'relop' 'num'   ')' '{' 'id' '=' 'num'    ';' '}' 'else' '{' STATEMENT '}'
```

/***Link to the parsing is added above ***/